On-the-Fly Topological Sort—A Basis for Interactive Debugging and Live Visualization of Parallel Programs.

Doug Kimelman*and Dror Zernik†

Abstract

This paper presents an optimal technique for on-the-fly ordering and matching of event data records that are being produced by a number of distinct processors. This is essential for effective interactive debugging and live visualization of parallel programs. The technique involves on-the-fly construction of the causality graph of the execution of the program. A sliding window over the graph is maintained by discarding portions of the graph as soon as they are no longer required for ensuring correct order of subsequent program events. The sort places an event record into the causality graph when it is received, places it into the output stream as soon as possible—as soon as all of its predecessors in the causal order have been placed into the output, and discards the event record as soon as possible—as soon as all of its successors in the causal order notice that it has been output. This technique is optimal in terms of the amount of space required for the sort, and in terms of the amount of additional delay incurred prior to delivery of an event record to its ultimate destination. We describe the algorithm in detail, show its optimality, and present results from experiments on a Meiko system running with PICL and ParaGraph.

1 Introduction: Event Ordering and Correlation

A continuous stream of event data describing the progress of program execution is often required for either interactive debugging or live visualization of parallel programs. This stream is essential for recognizing interesting points in program execution, which allows breakpointing; and it is essential for tracking changes in program state, which allows dynamic display of program behavior. The event data stream is most effective if the individual event records are in order by logical time, or "causality", and if event records concerning corresponding events on various processors have been correlated. For example, the event records for receipt of a broadcast message should not precede the event record for transmission of the broadcast message, and the time at which a message was sent could be added to the event record for receipt of the message.

Unfortunately, it is often the case that a separate stream is produced independently by each processor of the system without the benefit of a global clock. As well, there are often different degrees of buffering and latency involved in delivery of the various streams to some central point.

This paper presents a technique for on-the-fly ordering and matching of events from a number of independently produced event data streams in order to produce a single coherent output stream. The technique is shown to be optimal in terms of the amount of space required for the sort, and in terms of the amount of additional delay incurred prior to delivery of the event records to their ultimate destination.

The remainder of this section discusses the motivation for this work in greater detail. Section 2 presents the algorithm for ordering and matching events, and section 3 describes experience with an implementation of the algorithm for the Meiko topology-reconfigurable transputer-based parallel machine.

1.1 Interactive Debugging

Interactive debugging involves stopping program execution at interesting points (and then examining and possibly altering state, before resuming execution). For sequential programs, "interesting point" is most frequently characterized quite simply as: flow of control past some location, or perhaps: access to a particular piece of data.

*IBM Thomas J. Watson Research Center, Yorktown Heights, NY. dnk@watson.ibm.com
†Electrical Engineering Faculty, Technion, Haifa, Israel. dror@ee.technion.ac.il

For a parallel or distributed system, state is much more complex, and "interesting point" is much more difficult to characterize. Typically, interesting points are defined in terms of events which occur during program execution by the various processors [19, 1, 9, 16, 4, 3].

Recognition of interesting points may be based on analysis performed at the various processors by code placed there for that purpose, or it may be based on analysis performed at some central location external to the collection of processors. In cases where analysis is external, the various processors must forward event data to the central location in order to provide notification of event occurrence.

It is often the case that mechanisms involving code that is embedded on the various processors, and that must communicate with counterpart code on other processors in order to recognize interesting points, are not practical in a given environment, or are not sufficiently general or powerful. Further, with newer generations of scalable highly-parallel systems [18, 8, 7], trace facilities for collecting event data and forwarding it to a central location are becoming common. These facilities will increasingly be considered for use in interactive debugging, and on-the-fly ordering and matching of the resulting event data streams will assume increasing importance.

1.2 Live Program Visualization

Program visualization often includes dynamic displays of program behavior. The displays are driven by tracking interesting changes in the state of a system as it executes a program. For a parallel system, these changes in state may arise out of a combination of events occurring on a number of different processors. It is most often the case, for purposes of visualization, that each processor of a parallel system delivers a stream of data records concerning these events to a central program visualization system.

Visualization may be performed "post-mortem"—after completion of program execution and after all of the event data has been gathered, as in [5, 14, 17], or visualization may be performed "live"—concurrent with program execution as event data arrives, as in [10, 2]. Live program visualization is essential for integration of debugging and visualization, as in [13]. The visualization system displays program behavior, as event data are received, while the debugger allows the program to advance from one breakpoint to the next during controlled replay, and allows detailed inspection of program state. As well, live visualization is preferable for monitoring the progress of long-running programs on large-scale systems, either to verify continued correct behavior of the program, or to determine the point at which problems arise. An essential component of live program visualization is on-the-fly ordering and matching of incoming event data.

1.3 Event Stream Collection

As event data records arrive at the central location, they must be ordered and correlated for purposes of debugging and visualization. Often, records from different processors arrive via different independent streams, and they must be passed on to the rest of the debugging or visualization system in a single stream.

Causal order, not just arrival order, must be preserved as the data is passed on. For example, receipt of a message must not be reported before transmission of the message. Without causal order, state analysis and display is meaningless. Further, event data must be passed on as soon as possible. With large delays between the occurrence of a high-level event and its recognition, use of mechanisms such as breakpointing, undo and replay becomes prohibitively expensive.

Correlation of data from events which arrive at different times is often necessary for purposes of visualization. For example, the time at which a message was sent must be available when receipt of the message is being processed. This is required in order to drive the ubiquitous display of processor timelines with superimposed interprocessor communication [12, 5].

Unfortunately, varying degrees of buffering and transmission delays are encountered in delivery of the event data from the various processors to the central location. In this case, even with globally consistent synchronized clocks, the approach commonly employed in post-mortem systems to ensure correct order (repeatedly choosing the oldest of the arrived events with respect to timestamps which were derived from the global clock) can fail, when the event which should be recognized as the oldest is missed because it is held up in transit.

Continual transmission of event data, and on-the-fly techniques, *do* introduce greater overhead into a system. Nonetheless, they are warranted, as they facilitate interactive debugging and live program visualiza-

tion. Further, in some cases, such as during controlled replay, the overhead is not a concern. In other cases, such as for debugging communication libraries or operating system loaders for parallel machines, on-the-fly techniques are the only possible alternative because problems often leave the system in a state which makes gathering of buffered traces impossible.

2 On-The-Fly Topological Sorting

To provide an effective basis for interactive debugging and live visualization of parallel programs, ordering and correlation of arriving event data must be based on topological sorting that preserves the causal order of the events—that is, the order imposed by the semantics of the operations constituting the events.

2.1 Basic Algorithm

The basic algorithm involves on-the-fly construction of the causality graph of the execution of the program. A sliding window over the graph is maintained by discarding portions of the graph as soon as they are no longer required for ensuring correct order of subsequent program events.

The causal order is defined by two rules: an event is an immediate predecessor in the causal order to the event which occurred next on its processor; and, a communication event which corresponds to data being sent to other processors is an immediate predecessor in the causal order to the events corresponding to receipt of the data by the other processors.

For non-blocking communication, this means that the beginning of a send operation is a predecessor to the end of the receive. For a representative selection of collective communication operations, as in [6]: the beginning of a group multicast operation, on the group member sending the data, must precede the end of the operation on all other members; the end of a combine or concatenate operation, on the group member receiving the data, must follow the beginning of the operation on all other members; the end of a prefix or shift operation, for each group member, must follow the beginning of the operation on its preceding neighbor; and, the end of an exchange operation, on both members of a pair, must follow the beginning of the operation on the other member.

Figure 1 provides an overview of the data maintained for a node in the graph and Figure 2 provides an overview of the basic algorithm. An event is referred to as *observed* when it is placed into the graph, and it is referred to as *reported* when it is placed into the output. The sort repeatedly takes an event record as input (line 1), and places it into the graph as a node (line 3) according to the causal order—edges connect it to each of its immediate predecessors and successors. The sort then takes any actions which have become possible because of the presence of the new node. An event record is placed into the output stream (line 16) as soon as possible—as soon as all of its predecessors in the causal order have been placed into the output (checked at line 14), and an event record is deleted from the graph (line 13) as soon as possible—as soon as all of its successors in the causal order notice that it has been output (checked at line 11 — note that if a successor is present, it *has* noticed, either when the successor entered the graph and counted its reported predecessors at line 4, or when this node was output, via line 19).

2.2 Assumptions

Assumptions concerning the input streams of event data are: that event delivery from any processor to the central location is FIFO (although this is not necessary if each processor sequence-numbers the events which it generates); and that, even for collective communication among dynamically formed groups of processors, it is always possible to know from the contents of an event record, and all which preceded it, the number of predecessors and successors that it has in the causal order, and their identities. This seems quite reasonable in view of the typical contents of current system traces, and recent trends towards deterministic constructs in loosely synchronous and data-parallel programming.

Note that only the *number* of predecessors and successors expected for a node is stored; identifiers are not. The identifiers are only required when a node is first inserted into the graph, in order to connect the node to its predecessors and successors that are already present in the graph. Subsequently, a node already in the graph requires no record of the identifiers. As its other predecessors and successors arrive in the graph,

```
event data;
number of successors expected;
number of predecessors expected;
graph structure {
   list of predecessors;
   list of successors;
   number of successors;
}
number of predecessors reported;
boolean: this node has been reported;
```

Figure 1: Data maintained for a graph node.

```
   main
1     LOOP reading events
2        build node from event;
3        insert node into graph;
4        count and save the
5           number of predecessors reported
6           for this node;
7        FOR each predecessor of this node
8           check_delete(predecessor);
9        check_report(this node);

   check_delete(node)
10    IF node has been reported AND
11       number of successors of node ==
12       number of successors expected THEN
13          delete node from graph;

   check_report(node)
14    IF number of predecessors reported for node ==
15       number of predecessors expected THEN
16          do_report(node);

   do_report(node)
17    report node;
18    set boolean reported for node;
19    FOR each successor of node
20          increment its number of predecessors reported;
21          check_report(successor);
22    check_delete(node);
```

Figure 2: Basic algorithm.

the connections to this node will occur "automatically" as part of connecting the arriving nodes to all of their predecessors and successors.

2.3 Correctness

Correctness of the algorithm requires that:

- *an event is* **reported** *only after all of its predecessors.*

This requirement is satisfied as follows:

- The algorithm only reports an event when its number of predecessors reported is equal to its number of predecessors expected (line 14). Further, the record of the number of predecessors reported is always correct:
  - Upon entry of a node into the graph, the algorithm finds and counts all of the node's predecessors that have already been reported (line 4).
  - Subsequently, whenever another of the node's predecessors is reported, the count for this node is updated (line 20).

Thus the algorithm is correct.

2.4 Optimality

Optimality of the algorithm requires that:

1. *a node is* **reported** *as soon as possible*
2. *a node is* **deleted** *as soon as possible*

These requirements are satisfied as follows:

1. There are two possibilities for the point in time at which to report a node:
   (a) immediately upon entry to the graph
   — upon entry, the algorithm counts predecessors (line 4) and then checks if all predecessors have been reported (line 14)
   (b) or, if not upon entry to the graph, which can only occur because a predecessor has not been reported yet, then upon finally reporting the last of the predecessors of this node
   — each time one of a node's predecessors is reported, the algorithm checks (line 21) to see if the node can now be reported

   Thus the first requirement is satisfied.

2. There are two possibilities for the point in time at which to delete a node:
   (a) as soon as it is reported (which was shown above to happen as soon as possible)
   — as soon as a node is reported, the algorithm checks (line 22) whether the node can be deleted
   (b) or, if not as soon as it is reported, which can only occur because a successor has yet to arrive in the graph and notice that this node has been reported, then upon arrival of the last of the successors of this node
   — each time another successor of a node arrives in the graph and takes note of all of its predecessors, the algorithm checks (line 8) whether the predecessor node can be deleted

   Thus the second requirement is satisfied.

Thus the algorithm is optimal.

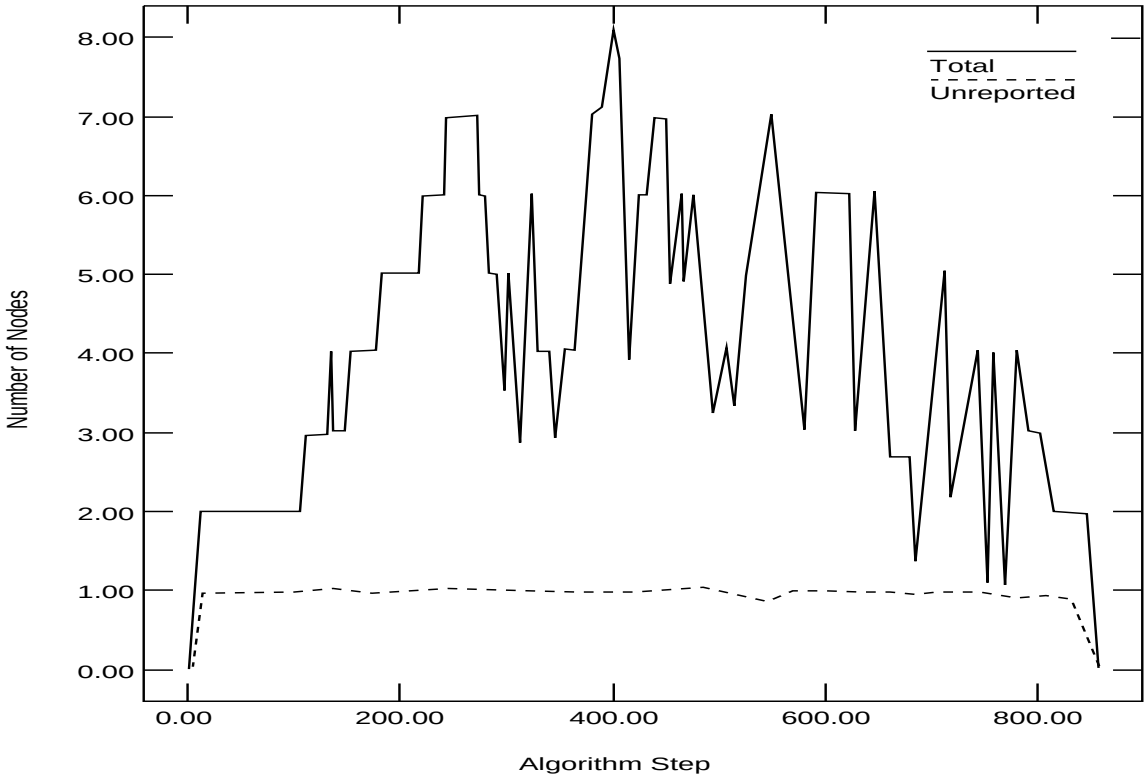

Figure 3: Size of the causality graph window over time.

2.5 Complexity

Complexity of the algorithm is $O(n * d)$ time and $O(n)$ memory, where n is the number of events observed and d is the maximal degree of a node in the causality graph. The time complexity is clear, since over the course of a run any such algorithm must always process every event and, for each event, this algorithm visits every predecessor (lines 4 and 7) and every successor (line 19) present in the graph. The space complexity arises from the fact that, in the worst case, no nodes are deleted from the graph until the end of the run, after all nodes have arrived in the graph.

A far more meaningful measure for such algorithms is *typical* space requirements, and average delay in reporting an event after it is observed. This is determined by the combination of asynchrony in the application, asynchrony in the message delivery medium, and asynchrony in the event data delivery medium. These contribute to nodes being held in the graph, prior to deletion, waiting for corresponding events to be reported. In actual practice, space requirements of this algorithm are very small — typically a few nodes per processor. This means that in most cases an event is reported and deleted very soon after it is observed. In particular, typical space requirements are *not* proportional to the size of the trace. Figure 3 shows the size of the graph over time for a sample run, and the next section describes in greater detail experience with an implementation of the algorithm.

It is interesting to note that the size of the graph over time is a reflection of the degree of asynchrony apparent to the user. Further, for applications which do not exhibit a large degree of asynchrony, the points in time where the graph grows often correspond to the points where messages would appear to be received before they are sent, for schemes which do not preserve causal order.

2.6 Timestamp Adjustment and Event Data Correlation

The algorithm also adjusts timestamps in order to compensate for local clock drift, in a manner similar to [11], but external to the system being monitored.

Timestamp adjustment is based on a record of accumulated clock drift maintained for each processor. The final timestamp for a reported event is the sum of its incoming timestamp and the drift of its processor.

For each node, a record is kept of the maximum accumulated drift required of its processor in order to keep it later than any of its predecessors. That is, whenever a successor notices that a predecessor has been reported, it subtracts its own timestamp from the final timestamp of its predecessor, and keeps the maximum across all of its predecessors. When a node is reported, the accumulated drift of its processor is increased to the maximum that the node requires, if the processor drift was not at least that large already.

Just prior to reporting an event, any correlation which is required may take place. For example, for purposes of driving the display of processor timelines and messages passing between them, the timestamp of an event corresponding to transmission of a message can be appended to the records of events corresponding to receipt of the message by other processors. Note that timestamps may only be copied from predecessors to successors. The timestamp of the successor is not yet final, and thus may not yet be added to the predecessor record; but the event record for the predecessor must not be detained because it must be reported as soon as is possible with respect to the causal order.

3 Experience Monitoring a Meiko Parallel System

For purposes of experimentation and evaluation, on-the-fly topological sorting was implemented as the front-end for live monitoring of a Meiko topology-reconfigurable parallel system [15] running with PICL and ParaGraph [5] as well as Visage [20]. To achieve continual delivery of event data, the PICL buffer size was reduced to a minimum. For purposes of comparison, a timestamp-based merge also was implemented.

The application being monitored was one in which a vector was summed repeatedly using a pipelined algorithm implemented with the machine configured as a ring.

Without the topological sort, a large number of messages appeared to have traveled backwards in time—that is, they appeared to have been received before they were sent. This was due to variations in delivery latency for event data. Further, delays were incurred due to the fact that the timestamp-based merge had to wait for at least one event to be present from each processor before it could proceed. In an effort to improve the situation, buffers were flushed periodically regardless of whether or not they were full, and *heartbeat* events were generated if the buffer was empty. With a PICL buffer size of 100 bytes, between 10 and 20 percent of messages still appeared to have traveled backwards in time.

Use of the sorting algorithm corrected the problem. Messages never appeared to have traveled backwards in time, and no time was lost waiting for timeouts to generate heartbeats on processors which were not actively communicating with others.

Figure 3 shows the size of the causality graph over time for this experiment. Both the total number of nodes in the graph and the number of nodes which have been observed but have yet to be reported are plotted at each step of the algorithm—that is, at each point in time where a node has been entered into the graph and all actions which have become possible as a result of the entry have been taken. Thus the area below the "Unreported" curve corresponds to nodes which are waiting for a predecessor to be reported before they can be reported, and the area above the curve corresponds to nodes which have been reported but which are waiting for successors to notice before they can be deleted. The average number of nodes in the graph over the course of the run was 3.6[1] and the average number of unreported nodes was 0.6. Thus the additional latency introduced by this algorithm between the time an event is delivered and the time it is reported is quite small on average.

4 Conclusion

The algorithm presented here is an optimal technique for on-the-fly ordering and correlation of event data external to the system being monitored. It handles collective communication, does not require a global clock, preserves causal order, adjusts timestamps derived from local clocks, and provides for correlation of data between records concerning corresponding events. Further, the algorithm is robust in the presence of varying

[1] A number of downward spikes in the "Total" curve do not appear as deep as they should due to compression of the plot to fit on a standard-sized page.

degrees of buffering and transmission delays in the delivery of event data from the application processors to their ultimate destination.

With this technique, it is realistic for parallel systems developers to expect to be able to construct effective tools for interactive debugging and live program visualization, even where no special support exists for a global clock.

References

[1] P. Bates. "Debugging Heterogeneous Distributed Systems Using Event-Based Models of Behavior," *Proc. ACM Workshop on Parallel and Distributed Debugging*, *ACM SIGPLAN Notices* 24(1):11–22, Jan. 1989.

[2] T. Bemmerl and P. Braun. "Visualization of Message Passing Parallel Programs," *Proc. CONPAR 92*, *Lecture Notes in Computer Science* 634:79–90, Springer-Verlag, 1992.

[3] B. Bruegge. "A Portable Platform for Distributed Event Environments," *Proc. ACM/ONR Workshop on Parallel and Distributed Debugging*, *ACM SIGPLAN Notices* 26(12):184–193, Dec. 1991.

[4] Z. Aral and I. Gertner. "High-Level Debugging in Parasight," *Proc. ACM Workshop on Parallel and Distributed Debugging*, *ACM SIGPLAN Notices* 24(1):151–162, Jan. 1989.

[5] M.T. Heath and J.A. Etheridge. "Visualizing the Performance of Parallel Programs," *IEEE Software* 8(5):29–39, Sep. 1991.

[6] F.D. Bryant, H. Ho, P. de Jong, R. Lawrence and M. Snir. "An external user interface for scalable parallel systems," *IBM Research Center Technical Report*, Jun. 1992.

[7] IBM Corp. "IBM 9076 Scalable POWERparallel 1 — General Information," GH26-7219-0, Feb. 1993.

[8] Intel Supercomputer Systems Division. "Paragon XP/S Product Overview," Nov. 1991.

[9] M.K. Pongami, W. Hseush and G.E. Kaiser. "Debugging Multithreaded Programs with MPD," *IEEE Software* 8(3):37–43, May 1991.

[10] D.N. Kimelman and T.A. Ngo. "The RP3 Program Visualization Environment," *The IBM Journal of Research and Development* 35(6):635–651, Nov. 1991.

[11] L. Lamport. "Time, Clocks and the Ordering of Events in a Distributed System," *Communications of the ACM* 21(7):558–565, Jul. 1978.

[12] T.J. LeBlanc, J.M. Mellor-Crummey, and R.J. Fowler. "Analyzing Parallel Program Execution Using Multiple Views," *Journal of Parallel and Distributed Computing* 9(2):203–217, Jun. 1990.

[13] E. Leu and A. Schiper. "Execution Replay: a Mechanism for Integrating a Visualization Tool with a Symbolic Debugger," *Proc. CONPAR 92*, *Lecture Notes in Computer Science* 634:55–66, Springer-Verlag, 1992.

[14] A.D. Malony, D.H. Hammerslag and D.J. Jablonowski. "Traceview: A Trace Visualization Tool," *IEEE Software* 8(5):19–28, Sep. 1991.

[15] Meiko. "Computing Surface; SunOS CSTools," 83-MS020.

[16] B.P. Miller and J.D. Choi. "Breakpoints and Halting in Distributed Programs," *Proc. 8th Conference on Distributed Computing Systems*, pp. 316–323, 1988.

[17] D.A. Reed, R.D. Olson, R.A. Aydt, T.M. Madhyastha, T. Birkett, D.W. Jensen, B.A.A. Nazief and B.K. Totty. "Scalable Performance Environments for Parallel Systems," *University of Illinois Technical Report* UIUCDCS-R-91-1673, Mar. 1991.

[18] Thinking Machines Corp. "CM-5 Technical Summary," Oct. 1991.

[19] L. Rudolph, R.V. Rubin and D. Zernik. "Debugging Parallel Programs in Parallel," *Proc. ACM Workshop on Parallel and Distributed Debugging, ACM SIGPLAN Notices* 24(1):100–124, Jan. 1989.

[20] A. Rudich, D. Zernik and G. Zodik. "Visage - Visualization Graph Attributes - A Foundation For Parallel Programming Environments," *Proc. CNRS-NSF Collaboration Workshop on Environments and Tools for Parallel Scientific Computing*, J. Dongarra and B. Tourancheau, editors, Elsevier Science Publishers, Sep. 1992.